“Randomized Speed-ups in Parallel Computation”
by
Uzi Vishkin

Ultracomputer Note #66
Technical Report #107
February, 1984

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012

"Randomized Speed-ups in Parallel Computation"

by

Uzi Vishkin

Ultracomputer Note #66
Technical Report #107
February, 1984

This work was supported in part by DOE Grant #DE-AC02-76ER03077 and in part by NSF Grant No. NSF-MCS79-21258.

To be presented at the 16th STOC.

Randomized Speed-ups in Parallel Computation

(February 1984)

Uzi Vishkin

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

251 Mercer St., New York, NY 10012

Ultra Computer Note 66
Technical Report 107

ABSTRACT

The following problem is considered: given a linked list of length n, compute the distance of each element of the linked list from the end of the list. The problem has two standard deterministic algorithms: a linear time serial algorithm, and an $O((n\log n)/p + \log n)$ time parallel algorithm using p processors. A known conjecture states that it is impossible to design an $O(\log n)$ time deterministic parallel algorithm that uses only $n/\log n$ processors.

We present three randomized parallel algorithms for the problem. One of these algorithms runs almost-surely in time of $O(n/p + \log n\log^{*}n)$ using p processors on an exclusive-read exclusive-write parallel RAM.

*This research was supported by DOE grant DE-AC02-76ER03077 and by NSF grant NSF-MCS79-21258.

**To be presented at the 16th STOC.

I. Introduction

The family of models of computation used in this paper is the parallel random-access-machines (PRAMs). All members of this family employ p synchronous processors all having access to a common memory. The PRAM family has 3 notable members. In a concurrent-read concurrent-write (CRCW) PRAM simultaneous reading from the same memory location is allowed as well as simultaneous writing. In the latter case the lowest numbered processor succeeds. A concurrent-read exclusive-write (CREW) PRAM allows simultaneous reading into the same memory location but not simultaneous writing. An EREW PRAM does not allow simultaneous reading or writing. See [Vi-83a] for a recent survey of results concerning the PRAM family.

Let Seq(n) be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem being considered. Obviously, the best upper bound on the parallel time achievable using p processors without improving the sequential result is of the form O(Seq(n)/p). A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal*. An ideal goal for serial computation is to design linear time algorithms (O(n) time) An analogous ideal goal for parallel computation is to design algorithms whose running time is proportional to n/p , where p is the number of processors used. In this case we say that a parallel algorithm achieves *parallel linear* running time.

The following problem is considered. (See also Fig. 1).

Input. A linked list of length n. It is given in an array of length n, not necessarily in the order of the linked list. Each of the n elements (except the last element in the linked list) has a pointer to its subsequent element in the linked list.

The list-ranking problem. Compute for each element its distance, counting elements, from the end of the list.

The problem has a trivial linear time serial algorithm. However, Wyllie [W-79] conjectured that $\Omega(n)$ processors are required in order to get $O(\log n)$ time. If true this implies, in particular, that there is no optimal speed-up parallel algorithm for $n/\log n$ processors. It is further conjectured that optimal speed-up is impossible even for less than $n/\log n$ processors (we leave open for how much less).

The goal of this paper is to relax the gap between the apparently most efficient deterministic parallel algorithm and the ideal goal of optimal speed-up by randomized parallel algorithms. Our algorithms obtain the running times mentioned below with probability that converges rapidly to one as n grows. Our strongest results are:

(1) A parallel algorithm that runs in $O(n/p)$ time using $p \leq n/(\log n \log^* n)$ processors on an EREW PRAM. (Observe that this algorithm achieves optimal speed-up). Recall that $\log^* n$ grows extremely slow and can be viewed as a constant for all practical purposes. (For instance, $\log^* 2^{65536} = 5$. See the function G in [AHU-74], p. 133). In particular, it runs in "about" $O(\log n)$ time using "about" $n/\log n$ processors.

(2) An $O(\log n)$ time algorithm using $n\log\log n/\log n$ processors on a CRCW PRAM.

The list of optimal speed-up parallel algorithms obtained so far is fairly short in spite of the interest in them. Let us mention the few known parallel linear algorithms: computaion of partial "sums" of n variables, where the word "sum" stands for any associative binary operation (this obvious algorithm is stated in the next section), [SV-81] for finding the maximum among n elements and merging, [Vi-83b] for finding the k smallest out of n elements, [G-84] for string matching, [CLC-81] and [Vi-81] for computing connected components of dense graphs, [TC-82] and [TV-83] for computing biconnected components of dense graphs and [BV-83] for generation of a computation tree form of an arithmetic expression and for finding matches in a sequence of parentheses. In addition, there are optimal speed-up algorithms for two more problems: [AKS-83] and [SV-81] for sorting and [PVW-83] for various operations on 2-3 tree.

Finding examples where randomized algorithms apparently beat the performance of their best possible deterministic counterparts is considered an interesting question in computational complexity. There are only a few examples where randomization is proven to be complexity effective. See, for instance, [Ra-83] and [MV-83]. See [Ra-76] for more on this concept of randomization.

Randomization in parallel computation. [Re-81] gave a randomized algorithm for selection of the k smallest element out of a set of n elements in a decision tree model of parallel computation. It runs in O(1) time using n processors with probabitily that converges rapidly to one. [Me-82] showed that it is possible to use similar ideas to [Re-81] and [SV-81] in order to get in the CRCW PRAM an O(1) time algorithm for finding the maximum among n elements using n processors with a similar probability. This is particularly interesting since [Va-75] proved that n processors need $\Omega(\log\log n)$ time in a parallel comparison model of computation in order to find the maximum among n elements. The selection algorithm of [Re-81] can be implemented (in a straightforward manner) to run on the EREW PRAM in $O(n/p)$ time using $p \leq n/\log n$ processors with a similar probability. We do not know if a deterministic algorithm can match this result. (The deterministic selection algorithm of [Vi-83b] runs in time $O(n/p)$ time using $p \leq n/(\log n\log\log n)$ processors). See also [RV-83] for a randomized sorting algorithm that involve parallel processors.

In the paper we actually present three algorithms. While the deterministic parts of all these algorithms are similar they differ considerably in the way in which randomization is applied. This facilitates a comparative demonstration of the role of randomization in this instance of parallel computation; i.e., we can focus on the contribution of the specific way in which randomization is applied in each of these algorithms. The simplicity of the definition of our problem helps also in this direction. We believe that randomization will become an important tool in the design of efficient parallel algorithms.

The list ranking problem is encountered often in the design of parallel algorithms. For instance, some of the tree procedures in the biconnectivity algorithm of [TV-83] may have now the same efficiency as the randomized algorithms presented here.

Remark. [CSV-82] classified many problems with respect to how fast they can be solved by CRCW PRAM algorithms using a polynomial number of processors. We show a reduction from sorting into the list ranking problem that preserves time up to a constant factor and number of processors up to a polynomial. Say that we wish to sort an array of n numbers A(1),A(2),...,A(n). For each A(i) compute in O(1) time the largest number which is smaller than A(i) using the constant time algorithm of [SV-81] for finding the maximum among n elements. This results in a linked list that has to be "ranked" as required in our problem. We do not know if a reduction in the reverse direction exists. Moreover, [CSV-82] give an O(log n/loglog n) time algorithm for sorting using a polynomial number of processors, but we do not know whether such an upper bound can be obtained for the list ranking problem.

Among other things, the next section recollects an optimal speed-up deterministic parallel algorithm that uses balanced trees. It is used later for two purposes: (1) as a subroutine, and (2) for explanation of the randomized parallel algorithms. A careful look shows that all our algorithms essentially manipulate randomization into the (sometimes remote) framework of this algorithm.

A randomized parallel algorithm whose running time is $O(\log n \log\log n)$ (with probability that converges rapidly to one) using $n/(\log n \log\log n)$ processors is presented in Sec. 3. The algorithm given in Sec. 4 improves this result. Its running time is $O(\log n \log^* n)$ (with probability that converges rapidly to one) using $n/(\log n \log^* n)$ processors. An $O(\log n)$ time algorithm (with probability that converges to one) using $n \log\log n/\log n$ processors is presented in Sec. 5.

II. Preliminaries

Theorem (Brent). Any synchronous parallel algorithm of time t that consists of a total of x elementary operations can be implemented by p processors within a time of $\lceil x/p \rceil + t$.

Proof of Brent's theorem. Let x_i denote the number of operations performed by the algorithm in time i $\left(\sum_1^t x_i = x\right)$. We now use the p processors to "simulate" the algorithm. Since all the operations in time i can be executed simultaneously, they can be computed by the p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time of

$$\sum_1^t \lceil x_i/p \rceil \leq \sum_1^t (x_i/p + 1) \leq \lceil x/p \rceil + t . \qquad \circ$$

Remark. The proof of Brent's theorem poses two implementation problems. The first is to evaluate x_i at the beginning of time i in the algorithm. The second is to assign the processors to their jobs.

Recall the following standard deterministic parallel algorithm for the list-ranking problem (defined in the Introduction). Say that we have n processors. Assign a processor to each of the n elements. Denote the pointer of element i of the input array by D(i) and initialize $R(i) := 1$, $1 \leq i \leq n$. We set D(t) := "end of list" (where t is the last element in the linked list), D("end of list") := "end of list" and R("end of list") := 0 .

Apply $\lceil \log n \rceil$ iterations:

<u>for</u> processor i, $1 \le i \le n$, <u>pardo</u> (perform in parallel)

R(i) := R(i)+R(D(i)); D(i) := D(D(i)) (To be called the short-cut operation). (See Fig. 2)

Note that a total of $\Omega(n\log n)$ short-cuts is required in this algorithm. It runs in time of $O((n\log n)/p + \log n)$ using p processors on an EREW PRAM and solves the list ranking problem into the vector R.

<u>Implementation Remark 1.</u> In order to derive this running time from Brent's theorem n has to be broadcasted to all p processors. This takes additional $O(\log p)$ time.

<u>Proposition 1.</u> (This is a variant (due to [AV-79], p. 158) of Chernoffs' bounds) For all ℓ, q, β with $1 \le q \le 1$, $0 \le \beta \le 1$

$$\text{(a)} \quad \sum_{k=0}^{\lfloor (1-\beta)\ell q \rfloor} \binom{\ell}{k} q^k (1-q)^{\ell-k} \le \exp(-\beta^2 \ell q/2)$$

$$\text{(b)} \quad \sum_{k=\lceil (1+\beta)\ell q \rceil}^{\ell} \binom{\ell}{k} q^k (1-q)^{\ell-k} \le \exp(-\beta^2 \ell q/3)$$

Let X be a random variable having the hypergeometric distribution with parameters ℓ, Nq, N ($0 \le q \le 1$). One way to demonstrate this distribution is: we have a bag containing N balls Nq of them are white and the rest black. X denotes how many white balls we get while sampling ℓ balls without replacements. Let $L_{N,\ell,C}(q)$ be Prob($X \le C$) for any C.

Let Y be a random variable having the binomial distribution with parameters ℓ, q. In the above example Y denotes how many white balls we get while sampling ℓ balls with replacements. Let $L_{\ell,C}(q)$ be Prob($Y \le C$). See [F-50].

<u>Proposition 2.</u> (Due to Uhlmann, cf. [JK-69], p. 151).

(a) $L_{N,\ell,C}(q) - L_{\ell,C}(q) > 0$ for $0 < q \leq C(\ell-1)^{-1}N(N+1)^{-1}$

(b) $L_{N,\ell,C}(q) - L_{\ell,C}(q) < 0$ for $1 > q \geq (C(\ell-1)^{-1}N + 1)(N+1)^{-1}$

Balanced binary tree parallel algorithms.

One simple pattern of optimal speed-up deterministic parallel algorithms is the balanced binary tree. This pattern was used, among many others, by [W-79], [CLC-81] and [Vi-81]. Let us first demonstrate this pattern on the problems of computing sums and partial sums.

Input. An array of n numbers A(1),A(2),...,A(n). Assume, w.l.g. that $\log_2 n$ is an integer.

Problem. Compute their sum.

Algorithm. "Plant" a balanced binary tree with n leaves. Every node of the tree is denoted [h,j]. See Fig. 3. Leaf [0,j] corresponds to A(j). Associate a number B[h,j] with every node of the tree. Initialization.

```
for all $1 \leq j \leq n$ pardo B[0,j] := A(j).
for h := 1 to log n
for all $1 \leq j \leq 2^{\log n - h}$ pardo B[h,j] := B[h-1,2j-1] + B[h-1,2j].
```

B[log n,1] holds the desired sum.

Think first about an n processor implementation of this summation algorithm. It runs in O(log n) time. Then apply the proof of Brent's Theorem to get an alternative implementation that uses only n/log n processors and runs in O(log n) time. This summation algorithm can be extended to solve the following partial-sum problem.

Input. Same as for the summation problem.

Problem. Compute $\sum_{1}^{i} A(j)$ for all $1 \leq i \leq n$.

Algorithm. Perform the summation algorithm given above. An additional "down-sweep" of the tree (from the root to the leaves), which roughly amounts to reversing the operation of the summation algorithm, will complete the job: Associate another number $C[h,j]$ with each node $[h,j]$.

```
Initialization. $C[\log n,1] := 0$.
for $h := \log n-1$ downto $0$
  for all $1 \leq j \leq 2^{\log n - h}$ pardo if $j$ is odd then $C[h,j] := C[h+1,(j+1)/2]$
     else $C[h,j] := C[h+1,j/2] + B[h,j-1]$.
for all $1 \leq j \leq n$ pardo $C[0,j] := C[0,j] + B[0,j]$.
```

$C[0,j]$, $1 \leq j \leq n$, hold the desired partial-sums. This algorithm can also be implemented to run in $O(n/p + \log n)$ time using p processors on an EREW PRAM. (Apply Brent′s theorem and Implementation Remark 1.)

A wishful thinking. We want to find an algorithm for the list ranking problem whose total number of short-cuts is $O(n)$. If we could "plant" a balanced binary tree in our linked list (in the order of the linked list) it would have solved our problem: enter one at each leaf and apply the partial sum algorithm. A closer look at the summation part of such a partial sum computation reveals the following:

The operation of the for statement for h=1 corresponds to short-cuts at all even (relative to the linked list) locations. This results in a new linked list that connects only even locations of the original list, thereby, halving its length. Then, the for statement for h=2 corresponds to short-cuts at even locations of the new linked list and so on. See Fig. 4. To sum up: The for statement of the summation algorithm never performs a short-cut at two successive elements of the linked list at hand; and, therefore, the "input" to any operation of this for statement is a single linked list.

(Remark. The problem is of course that we do not know how to plant a balanced binary tree with respect to the linked list without actually solving first the list ranking problem itself. Since this "planting" needs the ranking mod 2, mod 4, mod 8,... as explained above).

In our randomized algorithms we plant "randomly balanced trees". That is, the short-cut operations are picked at random such that we never perform simultaneous short-cuts at two successive locations of the linked list. Thereby, we move iteratively from one linked list to another single shorter linked list.

III. The First Algorithm

The first algorithm forms the closest (among our algorithms) randomized analogy to the partial-sum algorithm. In particular, its first part is a randomized analogy to the summation algorithm. Its second part (Step 5 below) is similar to the extention of the summation algorithm to the partial-sum algorithm.

The algorithm uses $p \leq n/(\log n \log\log n)$ processors on an EREW PRAM.

Initialization. $m := n$. Each processor is assigned to a successive segment of length n/p in the input array. Similar to the deterministic algorithm denote the pointer of element i by $D(i)$ and initialize $R(i) := 1$, $1 \leq i \leq n$. We set $D(t) :=$ "end of list" (where t is the last element in the list), D("end of list") := "end of list" and R("end of list") := 0 .

```
while m > n/log n do
(Comment. The input to each iteration of this while loop is a linked list of
length m stored in an array of length m. The vector D contains for each element
its subsequent element in the linked list.)
Processor i, 1 ≤ i ≤ p, is assigned to segment [(i-1)m/p + 1,...,im/p] in the
array which forms the input to this while loop. (Assume w.l.g. that m/p is an
integer.)
for Processor i, 1 ≤ i ≤ p, pardo
```

Step 1. for j := (i-1)m/p + 1 to im/p do

Toss a coin; Assign the result of the coin to c(j); SURVIVE(j) := 1.

(Comments. Each coin gets 0 or 1 with probabilty one half. Assume that c("end of list") is always 0. SURVIVE(j) is initialized to 1. It implies that element j is included in the output linked list of this iteration of the while loop unless SURVIVE(j) is set to 0 in Step 2.)

Step 2. for j := (i-1)m/p + 1 to im/p do

for each element j such that c(j)=0 and c(D(j))=1 do

OP(i,t) := (D(j),j,R(j)); SURVIVE(D(j)) := 0;

R(j) := R(j) + R(D(j)); D(j) := D(D(j)) (shortcut).

(Comments. 1. The shortcut operation cannot be applied to two successive elements in the linked list. 2. Each element whose predecessor in the list performed a shortcut remains with no incoming pointers. It is "deleted" in Step 3. The instruction SURVIVE(D(j)) := 0 takes care of this. 3. The parameter t stands for the present time. The information in OP(i,t) enables us to reconstruct later the operation of processor i at time t. This is used in Step 5 to derive the final value of R(D(j)) from the final value of R(j)).

Step 3. Perform the balanced binary tree partial-sum computation described in the previous section with respect to the vector SURVIVE. As a result:

(1) $m := \sum_j$ SURVIVE(j), and

(2) each element j with SURVIVE(j)=1 gets its entry number in a (contracted) array of length m containing the output linked list.

(This array is the input for the next (if any) iteration of the while loop.)

od

Let T_1 (resp. T_2) be the first (resp. last) time unit for which an assignment into OP(,) was performed.

Step 4. Apply a simulation of the deterministic algorithm by p processors to the current array.

Step 5.

```
for Processor i, $1 \le i \le p$, pardo
for t := $T_2$ downto $T_1$ do
R(OP(i,t).1) := R(OP(i,t).2) - OP(i,t).3  .
```

(*Comment.* OP(i,t).k , k=1,2,3, represents the fields of OP(i,t)).

Implementation remark. Each time m gets a new value broadcast it to all processors as in Implementation Remark 1 of the previous section.

Complexity.

Theorem. The algorithm runs in time $O(n/p)$, with probabilty $1 - O(e^{-\Omega(n/\log n)})$, using $p \le n/(\log n\log\log n)$ processors.

Proof. Each iteration of the *while* loop takes a total of $O(m/p + \log m)$ time. Step 4 takes $O(n/p)$ time. The time for Step 5 is bounded by the time for the *while* loop.

Denote the length of the input array to iteration i of the *while* loop by m_i , for $i = 1,2,...$. The quality of our algorithm is determined by the actual values of the sequence $m_1, m_2, ...$.

Lemma. There exists a positive integer n_0 such that for all $n \ge n_0$ the following is satisfied: the probability that $m_i \le (15/16)^{i-1}n$, for all $i = 2,3,...$, is $1 - O(e^{-\Omega(n/\log n)})$.

In this case: (1) the number of iterations of the *while* loop is $O(\log\log n)$; and (2) the time spent on the *while* loop is

$$O\left(\sum_{1}^{O(\log\log n)} ((15/16)^i n/p + \log n)\right) = O(n/p + \log n\log\log n)$$

and the Theorem follows.

It remains to prove the Lemma.

Observation 1. Element j performs a shortcut in Step 2 with probability $1/4 = (\mathrm{Prob}(c(j))=0)(\mathrm{Prob}(c(D(j))=1))$. (Unless D(j) is "end of list").

Observation 2. Let i and j be two elements in even locations of the linked list. Then their probabilities for performing shortcuts in Step 2 are independent.

We show that with high probability sufficiently many shortcuts in even locations are performed. For this apply Proposition 1(a) of the previous section with $\ell=n/2$, $q=1/4$ and $\beta=1/2$. The probability that following one iteration $< n/8$ of the $n/2$ possible short-cuts in even location are performed is $\leq e^{-n/64}$. This bounds also the probability that m_2, the length of the list following one iteration, is $\geq 15n/16$. Let $i > 1$ be an integer. Suppose that for all $2 \leq j \leq i$ $n^j \leq (15/16)^{j-1}n$. By similar considerations the probability that $m_{i+1} \leq (15/16)^i n$ is $\geq 1 - e^{-m_i/64}$.

We already implied that the number of iterations in case the sequence of the m^i-s satisfy the bounds of the Lemma is $O(\log\log n)$. The probability that this will happen is $\geq (1-e^{-n/64\log n})^{O(\log\log n)} \geq 1 - O(e^{-n/64\log n}\log\log n)$. Obviously, there exists n_0 such that for all $n \geq n_0$ this probability is $\geq 1 - O(e^{-\Omega(n/\log n)})$. This converges very rapidly to 1 as n grows.

IV. The Second Algorithm

Throughout this section we use $p = n/(\log n\log^* n)$ processors on an EREW PRAM. In the algorithm below processors are assigned to elements through random permutations. We use the fact that with very high probabilty only few of the processors are assigned to subsequent elements.

Initialization. $m := n$. Vectors R and D are defined and initialized as in the first algorithm. Throughout this section we use also D^{-1}, the inverse of D (to form a doubly linked list). D^{-1} is initialized in $O(n/p)$ time in a straightforward manner.

<u>while</u> $m > c_1 n/\log n$ <u>do</u>

(<u>Comment</u>. $c_1 (>1)$ is some proper constant. We elaborate on how to select c_1 in our complexity analysis.)

<u>Step 1</u>. Take a random (precomputed) permutation σ of $1,2,\ldots,m$. (Assume, w.l.g., that m/p is an integer). Assign processor i, $1 \leq i \leq p$, to the segment of m/p elements $[(i-1)m/p+1,\ldots,im/p]$ in the domain of σ.

<u>Step 2</u>. Processor i, $1 \leq i \leq p$, scans its segment from left to right in m/p pulses. At pulse t, $1 \leq t \leq m/p$, processor i is at the $\sigma((i-1)m/p+t)$ element of the input array. Denote this element by $a_{i,t}$ and its predecessor (if exists) $D^{-1}(a_{i,t})$ by $b_{i,t}$. Processor i marks $a_{i,t}$ as "accessed at pulse t".

<u>Processor i at pulse t</u>.

<u>if</u> $D(a_{i,t})$ is marked as accessed at pulse t or $a_{i,t}$ is the tail of the list

<u>then</u> $\text{SURVIVE}(a_{i,t}) := 1$

<u>else</u> $OP(i,t) := (a_{i,t}, b_{i,t}, R(b_{i,t}))$; $R(b_{i,t}) := R(b_{i,t}) + R(a_{i,t})$; $\text{SURVIVE}(a_{i,t}) := 0$; $D(b_{i,t}) := D(a_{i,t})$ (shortcut); $D^{-1}(D(b_{i,t}) := b_{i,t}$.

(<u>Comment</u>. The only case where $D^{-1}(a_{i,t})$ does not exist is when $a_{i,t}$ is the tail of the list.)

(<u>Explanation</u>. If currently there is no other processor at the element ahead of $a_{i,t}$ a shortcut is performed. Note, that unlike the previous algorithms the shortcut detours $a_{i,t}$ itself rather than its successor. This change is in order to avoid the possibilty that $a_{i,t}$ was detoured in previous pulses).

<u>Step 3</u>. A contraction of the remaining linked list into an array is performed as in Step 3 of the first algorithm. Assign their number into m.

(Comment. Later we refer to the elements not in the remaining linked list as deleted.)

```
od
```

Let T_1 (resp. T_2) be the first (resp. last) time unit for which an assignment into OP(,) was performed.

Step 4. Apply a simulation of the deterministic algorithm by p processors to the $\leq c_1 n/\log n$ remaining elements.

Step 5.

```
for Processor i, $1 \leq i \leq p$, pardo
for t := $T_2$ downto $T_1$ do
R(OP(i,t).1) := R(OP(i,t).2) - OP(i,t).3 .
```

Implementation remarks. 1. Each time m gets a new value broadcast it to all processors as in Implementation Remark 1 of Section 2. 2. The actual series of values of m is itself a random variable and is not known in advance. Question. Do we have to store precomputed random permutations on $[1,2,\ldots,m]$ for every possible m? Answer. Store random permutations only for powers of 2 (for "sufficiently large" numbers). Given an m take a random permutation for $[1,2,\ldots,2^{\lceil \log m \rceil}]$ and "contract" it into a (random) permutation on $[1,2,\ldots,m]$ similar to the way in which the vector SURVIVE is used to contract the array containing the linked list. This will not affect time complexity by more than a constant factor.

Complexity.

Theorem. The algorithm runs in time $O(n/p)$ $(= O(\log n \log^* n))$, with probabilty $1 - O(e^{-\Omega(n/(\log n\log^* n)^2)})$.

Proof. Each iteration of the while loop takes a total of $O(m/p + \log m)$

time. Step 4 takes $O(n/p)$ time. The time for Step 5 is bounded by the time for the while loop.

Denote the lengths of the input arrays to iteration i of the while loop by m_i , $i = 1,2,\ldots$. The quality of our algorithm is determined by the actual values of the sequence $m_1,m_2,\ldots$.

Lemma 1. There exist a constant $c_1 > 1$ and a positive integer n_0 such that for all $n \geq n_0$ the following is satisfied: the probability that $m_i \leq c_1 p \log^{i-1}(n/p)$, for all $i = 2,3,\ldots$, is $\geq 1 - O(e^{-\Omega(n/(\log n \log^* n)^2)})$. (Where $\log^i$ is defined as follows: $\log^0$ is the empty string and $\log^i = \log \log^{i-1}$). In this case: (1) the number of iterations of the while loop is $\leq \log^* n$ (This follows readily from our assumption $p = n/(\log n \log^* n)$); and (2) the time spent on the while loop is

$$O\Big(\sum_{1}^{O(\log^* n)} ((p \log^{i-1}(n/p))/p + \log n)\Big)$$

The log n term of the summation contributes $O(\log n \log^* n)$ to the total. For $i=1$ the first element of the summation is $n/p = \log n \log^* n$. For $i \geq 2$ this term is $O(\log n)$, and therefore the total is $O(\log n \log^* n)$, and the Theorem follows. The rest of this section is devoted to prove Lemma 1.

An iteration i of the while loop starts with an array of $m(=m_i)$ elements $\{1,\ldots,m\}$ which forms a linked list. Till Corollary 1 below we consider only the first pulse of iteration i. In the first pulse a subset of size p $I_1 = \{\sigma(1),\sigma(m/p + 1),\ldots\}$ is selected. The selected subset can be partitioned into intervals as follows: two elements of the subset belong to the same interval if all the elements among them in the linked list belong to the subset. Let $R = R(m,p)$ be the random variable representing the number of intervals. Observe that having r intervals imply that the number of short-cuts performed in Step 2 is r or r-1 (remember the tail of the list case that happens once during an iteration). Thus, as a result of the first pulse the following elements are still in the linked list: the m-p elements that we did not "touch" plus $\leq$ p-r+1 elements (to be called "survivors") where a shortcut could have been but was not performed.

Lemma 2 gives the distribution of R(m,p).

Lemma 2. Prob(R(m,p)=r) the probabilty that R(m,p)=r, $1 \leq r \leq p$, is $\binom{p-1}{p-r}\binom{m-p+1}{r}/\binom{m}{p}$.

Proof. The linked list is being accessed through a random permutation. (All permutations are equally likely).

Claim. The number of permutations for which the p elements being accessed form r intervals in the linked list is $\binom{p-1}{p-r}p!\binom{m-p+1}{r}(m-p)!$

The claim together with the fact that the total number of permutations is m! imply Lemma 2.

Proof of claim. (a) The number of possibilities to partition the p elements into r non-empty intervals is $\binom{p-1}{r-1}p!$.

Let the p elements have p possible locations in a row. Put a "divider" between each successive pair of locations. Select r-1 of the p-1 dividers and sort the p elements into their possible locations.

(b) The number of possibilities to partition the m-p remaining elements into r+1 intervals such that the r-1 middle intervals are non-empty is $\binom{m-p+1}{r}(m-p)!$.

We look at m-p+1 pebbles in a row. Select r out of them. The length of the r+1 intervals is now identified as follows: The first (resp. the last) interval has the same number of elements as the number of pebbles to the left (resp. right) of the leftmost (resp. rightmost) selected pebble. The number of elements of the i-th interval, $2 \leq i \leq r$, is one plus the number of pebbles between the (i-1)-st and the i-th selected pebbles. It can be readily seen that this defines a 1-1 correspondence onto the set of such intervals. Finally, sort the m-p elements into their possible locations.

This completes the proofs of the claim and Lemma 2.

Lemma 2 shows that the distribution of $R(m,p)$ is actually hypergeometric.

Define another random variable $X = p - R(m,p)$. The distribution of X is hypergeometric with parameters $\ell=p$, $q=(p-1)/m$, $N=m$, $Nq=p-1$ (see Sec. 2) and therefore $E(X) = p(p-1)/m$ (see [F-50]).

Let Y be a random variable whose distribution is binomial with parameters $\ell=p$, $q=(p-1)/m$.

Lemma 3. $\mathrm{Prob}(Y \geq C) > \mathrm{Prob}(X \geq C)$ for $C \geq p(p-1)/m$.

Proof. In order to apply Proposition 2(a) (Sec. 2) we have to show that $q = (p-1)/m$ is less than or equal $C(\ell-1)^{-1}N(N+1)^{-1}$ which is $\geq (p(p-1)/m)(1/(p-1)(m/(m+1)) = (p/m)(m/(m+1))$. Since m is much greater than p , $p-1 \leq pm/(m+1)$ and the required inequality follows. ∘

Corollary 1. $\mathrm{Prob}(X \geq (3/2)p(p-1)/m) \leq e^{-p(p-1)/12m}$.

Proof. By Proposition 1(b) (Sec. 1), $\mathrm{Prob}(Y \geq (3/2)p(p-1)/m) \leq e^{-p(p-1)/12m}$ ∘

Remark. We selected arbitrarily $\beta=1/2$ for the application of Proposition 1(b). In the analysis it is possible to trade lower probabilty for success (smaller β) for a faster algorithm.

So we got an upper bound on the probabilty that the number of survivors of the first pulse of an iteration of the while loop exceeds $(3/2)p(p-1)/m + 1$ (recall that the number of survivors is $\leq p-r+1$).

Next we show how to apply this analysis of the first pulse to the pulses that follow. Let us start with the second pulse where another subset of size p of the linked list $I_2 = \{\sigma(2), \sigma(m/p+2), \ldots\}$ is selected. In order to simplify the analysis our definition of intervals does not take advantage of the fact that survivors of the first pulse may increase the number of actual intervals: Two elements of I_2 belong to the same interval if all the elements among them in the current linked list either belong to I_2 or are survivors of the first pulse. The main observation is that the number of intervals of the second pulse meets the distribution of $R(m-p,p)$. The reason for this is that after the first pulse we virtually delete p elements from the domain of the permutation σ $(\{1, m/p+1, \ldots\})$ as well as from the range of σ (the image of $\{1, m/p+1, \ldots\}$ - the set I_1). Thus, all possible permutations on the remaining m-p elements of the domain are equally likely once the mapping of the p elements $\{1, m/p+1, \ldots\}$ is fixed.

The analysis of the first pulse carries through and results in the following: The probabilty that the number of survivors of the second pulse satisfies $\geq (3/2)p(p-1)/(m-p) + 1$ is $\leq e^{-p(p-1)/12(m-p)}$.

The definition of intervals at the subsequent pulses will likewise not take advantage of survivors of their preceding pulses. The distribution of the number of intervals at pulse i, $1 \leq i \leq m/p$, meets the distribution of $R(m-(i-1)p,p)$. And, the probabilty that the number of survivor of the (i+1)-st pulse satisfies $\geq (3/2)p(p-1)/(m-ip) + 1$ is $\leq e^{-p(p-1)/12(m-ip)}$. (Note that our analysis fails completely in measuring the gain in the (m/p)-th pulse).

Summing up the number of survivors over the m/p pulses gives

$$\sum_{i=0}^{(m/p)-1} ((3/2)p(p-1)/(m-ip) + 1 \leq (3/2)(p-1)\left[\sum_{i=0}^{(m/p)-1} (1/((m/p)-i)\right] + m/p$$

$$\leq (3/2)(p-1)\lceil \log(m/p) \rceil + m/p \qquad (4.1)$$

The probabilty that this will be the number of survivors is $\geq (1 - e^{-p(p-1)/12m}) \ldots (1 - e^{-p(p-1)/12(m-ip)}) \ldots$

We may weaken slightly this bound by observing that this probabilty is $\geq 1 - \left(\sum_{i=0}^{(m/p)-1} e^{-p(p-1)/12(m-ip)}\right) \geq 1 - (m/p)e^{-p(p-1)/12m}$ (4.2).

Recall that $c_1 n/\log n \leq m \leq n$.

Corollary 2. We consider an iteration of the loop that starts with m elements. For any constant $c_2 > 3/2$ there exists n_1 such that for all $n \geq n_1$ the probabilty that this iteration results in $\leq c_2 p\log(m/p)$ survivors is $\geq 1 - O(e^{-\Omega(n/(\log n\log^* n)^2)})$.

The selection of c_1 for Lemma 1. We select c_1 such that the following assertion is satisfied. Let $s_i = c_1 p\overset{i-1}{\log} (n/p)$. Obviously, $m_1 \leq s_1$.

Assertion. If m_i , the input length for the i-th iteration of the loop, is $\leq s_i$ and $m_i > c_1 n/\text{lon}\ n$ (the repeat condition of the while loop) then $c_2 p\log(m_i/p) \leq s_{i+1}$.

We want that $c_2 p\log(m_i/p) \leq s_{i+1}$. Since $m_i \leq s_i$ it suffices that $c_2 p\log(c_1 p(\overset{i-1}{\log} (n/p))/p) \leq c_1 p\overset{i}{\log} (n/p)$ which is the same as $c_2(\log c_1 + \overset{i}{\log} (n/p)) \leq c_1\overset{i}{\log} (n/p)$ and $c_2\log c_1 \leq (c_1-c_2)\overset{i}{\log} (n/p)$ (4.3).

The repeat condition of the while loop shows that for $c_1 > c_2$ and sufficiently large n_o (4.3) is satisfied for all $n \geq n_0$. Observe that if c_1 is selected to be slightly greater than c_2 then a small increase in c_1 may cause a sharp decrease in n_0 . ○

We are ready to finish the proof of Lemma 1. It was already argued that if $m_i \leq s_i$, for all i=2,3... , then the number of iterations of the loop is $\leq \log^* n$. Therefore, Corollary 2 (and considerations that led to it) implies that the probabilty that $m_i \leq s_i$ for all i=2,3,..., is $\leq 1 - O((\log^* n)e^{-\Omega(n/(\log n\log^* n)^2)}) = 1 - O(e^{-\Omega(n/(\log n\log^* n)^2)})$

This completes the proof of Lemma 1.

Remark. Recall that the selection of c_2 (which is a parameter of the complexity analysis only) was arbitrary ($\geq 3/2$). Besides its dependence on c_2 the selection of c_1 is arbitrary as well. The selection of c_1 affects both the time distribution of the while loop and the time of Step 4. Our analysis was aimed to show that there exists a choice of c_1 for which the running time is almost surely $O(n/p)$ using p processors provided that n is sufficiently large. We did not intend to exhaust the various trade-offs possible as a result of alternative choices for c_1 (and c_2). We leave this to the interested reader.

V. The Third Algorithm

In this section the number of processors p is nlnln n/ln n on a CRCW PRAM. Initialization is as in the first algorithm. In addition, SURVIVE(j) is initialized to 1 for all $1 \leq j \leq n$.

```
for Processor i, $1 \leq i \leq p$, pardo
Step 1. Select a random number $N_i$ between 1 and n, ($Prob(N_i=j) = 1/n$ for
$1 \leq j \leq n$), and "mark" the element in location $N_i$ of the array. If several
processors select the same number only one marks the element and the others quit
till Step 3.
Step 2. COUNTER(i) := 0
while COUNTER(i) $\leq \lceil \ln n \rceil$ and $D(N_i)$ is not marked do
OP(i,t) := $(D(N_i),N_i,R(N_i))$; SURVIVE($D(N_i)$) := 0;
$R(N_i) := R(N_i) + R(D(N_i))$; $D(N_i) := D(D(N_i))$
COUNTER(i) := COUNTER(i) + 1
od
```

(Explanation. Starting with the selected element a processor shortcuts over its ln n subsequent elements in the linked list or till it hits an element that was selected by another processor. This portion of the linked list is called the chain of this processor. The result is that each processor replaces its chain by a pointer from the tail (the selected element) to the head of the chain.)

The three comments following Step 2 of the first algorithm apply here as well. Note that in the second assignment $R(D(N_i))$ is always 1.)

Let T_1 (resp. T_2) be the first (resp. last) time unit for which an assignment into OP(,) was performed.

Step 3. A contraction of the remaining linked list into an array is performed as in Step 3 of the first algorithm.

Step 4. Apply a simulation of the deterministic algorithm by p processors to the current array.

Step 5.

```
for Processor i, 1 ≤ i ≤ p, pardo
for t := $T_2$ downto $T_1$ do
R(OP(i,t).1) := R(OP(i,t).2) - OP(i,t).3 .
```

Implementation remark. A slightly weaker concurrent-write assumption (than the one given in the definition of the models in the introduction) suffices for this algorithm: In case more than one processor attempts to write into the same memory location one of these processors succeeds but we do not know in advance which. Such an assumption was used in [SV-82].

Complexity.

In Lemma 1 and corollaries 1 and 2 below the $\lceil \ln n \rceil$ tail elements of the original linked list are excluded.

Lemma 1. Let x be an element of the input array. The probabilty that none of the $\lceil \ln n \rceil$ elements that precede x in the linked list was selected by one of the (p=)nlnln n/ln n processors in Step 1 is $\leq 1/\ln n$.

Proof. The probabilty that these $\lceil \ln n \rceil$ elements do not include the element selected by processor i, $1 \leq i \leq p$, is $\leq (n-\ln n)/n$. Since the selection of numbers by distinct processors is independent the probabilty that the $\lceil \ln n \rceil$ list elements preceding x do not include a selected element is $\leq ((n-\ln n)/n)^p = [(1 - \ln n/n)^{n/\ln n}]^{\ln\ln n}$. Since the monotone increasing sequence $\{(1 - 1/n)^n\}$ converges to $1/e$ we get $\leq (1/e)^{\ln\ln n} = 1/\ln n$.

Corollary 1. The expected number of elements that none of the $\lceil \ln n \rceil$ elements preceding them in the linked list was selected by a processor is $\leq n/\ln n$.

Corollary 2. The number of elements that none of the $\lceil \ln n \rceil$ elements preceding them in the linked list was selected by a processor is $\leq n\ln\ln n/\ln n$ with probabilty $1 - 1/\ln\ln n$.

Proof. Let E(X) be the expectation of a random variable X that gets only nonnegative values. Then, in general, $\text{Prob}(X \geq tE(X)) \leq 1/t$. The number of elements that none of the $\lceil \ln n \rceil$ elements preceding them in the linked list was selected by a processor is always non-negative. Therefore, Corollary 2 follows from Corollary 1.

Theorem. The length of the linked list following Step 2 is $\leq p + p + \lceil \ln n \rceil$ $(= 2n\ln\ln n/\ln n + \lceil \ln n \rceil)$ with probabilty $1 - 1/\ln\ln n$.

Proof. The first p represents pointers from tails to heads in chains of processors. The second p is from Corollary 2. Finally, there are $\lceil \ln n \rceil$ tail elements that were not considered above.

Let us sum up the time spent at each step. Step 1: $O(n/p)$. Steps 2 and 5: $O(\ln n)$. Step 3: $O(\ln n)$. Step 4: $O(\ln n)$ with probabilty $1 - 1/\ln\ln n$. Thus, the Third algorithm runs in $O(\ln n)$ time with probabilty $1 - 1/\ln\ln n$ using $n\ln\ln n/\ln n$ processors.

Acknowledgement. Much gratitude to P. Spirakis for stimulating discussions and many helpful suggestions. Comments by N. Megiddo, L. Rudolph, E. Shamir and M. Sipser are also gratefully acknowledged.

References

[AHU-74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[AKS-83] M. Ajtai, J. Komlós, and E. Szemerédi, "An O(n log n) sorting network," *Combinatorica* 3,1 (1983), 1-19.

[AV-79] D. Angluin and L.G. Valiant, "Fast probabilistic algorithms for Hamiltonian circuits and matching", *JCSS* 18 (1979), 155-193.

[BV-83] I. Bar-On and U. Vishkin, "Optimal parallel generation of a computation tree form", TR 90, Dept. of Computer Science, Courant Institute, NYU, 1983.

[CLC-81] F.Y. Chin, J. Lam and I. Chen, "Optimal parallel algorithms for the connected component problems," *Proc. 1981 International Conf. on Parallel Processing* (1981), 170-175.

[CSV-82] A.K. Chandra, L.J. Stockmeyer and U. Vishkin, "A complexity theory for unbounded fan-in parallelism", *Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982, 1-13.

[F-50] W. Feller, An Introduction to Probabilty Theory and its Applications, V. 1, Wiley, New York, 1950.

[G-84] Z. Galil, "Optimal parallel algorithms for string matching", *Proc. 16th ACM Symp. on Theory of Computing*, 1984, to appear.

[JK-69] N.J. Johnson and S. Kotz, Discrete Distributions, Houghton Mifflin Company, Boston, MA, 1969.

[Me-82] N. Megiddo, "Parallel algorithms for finding the maximum and the median almost surely in constant-time", preprint, 1982.

[MV-83] K. Mehlhorn and U. Vishkin, "Granularity of parallel memories", TR 89, Dept of Computer Science, Courant Institute, NYU, 1983. For an abstract see Proc. of the 9th Workshop on Graphtheoretic Concepts in Computer Science (WG-83), Fachbereich Mathematik, Universitat Osnabruck, June 1983.

[PVW-83] W. Paul, U. Vishkin and H. Wagener, "parallel dictionaries on 2-3 trees", *Proc. 10th ICALP*, Lecture Notes in Computer Science 154, Springer-Verlag, 1983, 597-609.

[Ra-76] M.O. Rabin, "Probabilistic algorithms", Algorithms and Complexity, J.F. Traub, Editor, Academic Press (1976), 21-39.

[Ra-83] M.O. Rabin, "Randomized Byzantine Generals", *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (1983), 403-409.

[Re-81] R. Reischuk, "A fast probabilistic parallel sorting algorithm", *Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science* (1981), 212-219.

[RV-83] J.H. Reif and L.G. Valiant, "A logarithmic time sort for linear size networks", *Proc. 15th Annual ACM Symp. on Theory of Computing* (1983), 10-16.

[SV-81] Y. Shiloach and U. Vishkin, "Finding the maximum merging, and sorting in a parallel computation model," *J. Algorithms* 2 (1981), 88-102.

[SV-82] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm", *J. Algorithms* 3 (1982), 57-67.

[TC-82] Y.H. Tsin and F.Y. Chin, "Efficient parallel algorithms for a class of graph theoretic problems," Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1982.

[TV-83] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", TR 69, Dept. of Computer Science, Courant Institute, NYU, 1983.

[Va-75] L.G. Valiant, "Parallelism in comparison problems", *SIAM J. Comp.* 4(1975), 348-355.

[Vi-81] U. Vishkin, "An optimal parallel connectivity algorithm," Technical Report RC 9149, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 1981. To appear in *Discrete Applied Mathematics*.

[Vi-83a] U. Vishkin, "Synchronous parallel computation - a survey", TR 71, Dept of Computer Science, Courant Institute, NYU, 1983.

[Vi-83b] U. Vishkin, "An optimal parallel algorithm for selection", preprint, 1983.

[W-79] J.C. Wyllie, "The complexity of parallel computation," TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

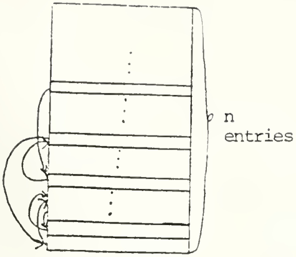

Fig.1.The input

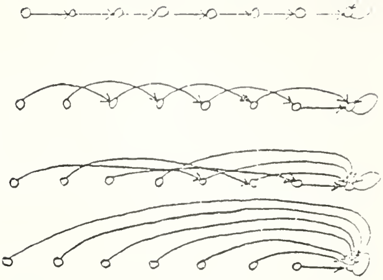

Fig.2.The deterministic algorithm

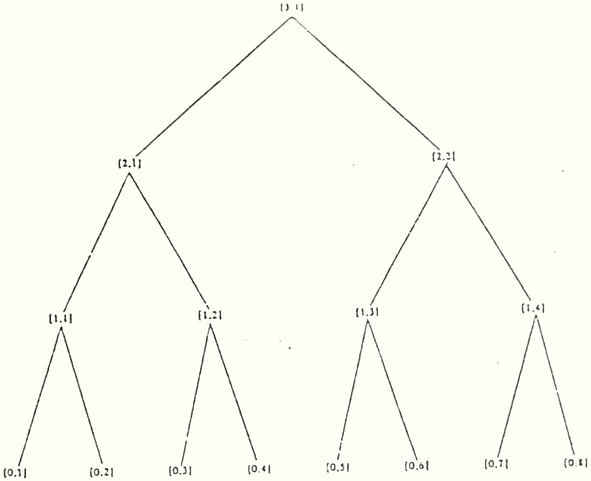

Fig.3. The balanced binary tree

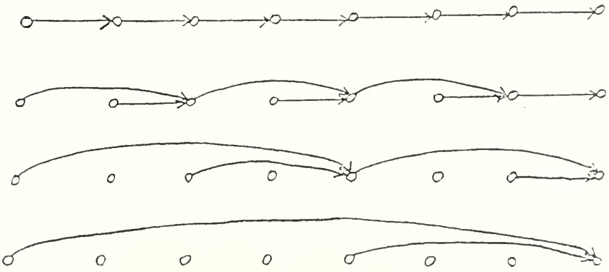

Fig.4. A "short-cut analogy" to the balanced binary tree algorithm.

NYU COMPSCI
TR-107
Vishkin
Randomized speed-ups in
parallel computation. c.2